Optimizing a Solver of Polymorphism Constraints: SEMI

Robert O'Callahan

June 1999

CMU-CS-99-136

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

As part of the Ajax system for analyzing Java bytecode programs, I have developed an analysis called SEMI, based on type inference with polymorphic recursion. SEMI has a number of optimizations which significantly decrease the time and space requirements for analyzing large programs. These optimizations exploit the characteristics of Java programs to make analysis tractable. These assumptions, and the optimizations that follow from them, may apply in other domains using type inference with polymorphic recursion. In this report, I describe the SEMI algorithm, the optimizations it incorporates, and the characteristics of Java programs that justify the optimizations.

This research is sponsored in part by the Defense Advanced Research Projects Agency and the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, F33615-93-1-1330, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0031 and in part by the National Science Foundation under Grant No. CCR-9523972. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The author was supported by a graduate fellowship from Microsoft Corporation.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency Rome Laboratory, the U.S. Government, or Microsoft Corporation.

Keywords: Java, bytecode, Ajax, polymorphism, polymorphic recursion, SEMI, constraint solving

1. Introduction

As part of the Ajax system for analyzing Java bytecode, I developed an analysis based on type inference with polymorphic recursion [H93], called SEMI. ("SEMI" stands for "semiunification".) SEMI has three major components:

- A "constraint generation" phase, which traverses the program code to build an initial constraint set
- A "solver" phase, responsible for analyzing the set of constraints and adding new constraints to the set in order to reach a "closed form"
- An "interpretation" phase, which examines the structure of the closed form to deduce information about the possible behaviors of the program.

This document focuses on the specification and implementation of the solver, in particular, the optimizations introduced to make it efficient. This document is a technical memorandum describing a variety of implementation techniques, and will mainly be of interest to someone implementing a system using similar technology. You have been warned.

An important point in the design and implementation of SEMI is that it is not, strictly speaking, a type inference system. I make no attempt to assign semantics to the structures produced by SEMI, except for one specific soundness theorem that relates properties of a "closed" constraint set to the behavior of the program which induced the initial set of constraints. The design is motivated by types, and it is often helpful to use concepts of type inference to think about SEMI, but the semantics usually associated with types do not apply.

2. The Constraint System

2.1. Type inference with polymorphic recursion

Traditional type inference with polymorphic recursion has been described as a solution procedure over constraints between terms [H93]. The terms represent types in a standard way; for example, there are unary symbols representing scalar types, a binary symbol representing function types, and variable-arity symbols representing tuple types. A term may also be a type variable, which may become bound to other type variables or terms. The constraints are of the form "$t_1 \preccurlyeq_i t_2$", meaning that t_2 is instance i of t_1. For example, t_1 may represent the type of a polymorphic function, and t_2 an instance of t_1 at some call site i.

The type inference procedure finds assignments to the type variables, and for each instance i a substitution S_i such that $t_1 \preccurlyeq_i t_2$ implies $S_i(t_1) = t_2$. Each substitution S_i describes how polymorphic type variables are instantiated at instance number i. Upon termination, the type inference procedure will return a *principal type* [DM82], i.e. the most general type assignment possible; if such a solution does not exist, it will signal an error. Termination is not guaranteed, but folklore asserts that the algorithm terminates for practical examples.

Unfortunately, terms are not well suited to representing recursive structures. Adding recursion to terms requires the introduction of a new type, e.g. "μt.T" where t occurs free in T, meaning the solution to the fixpoint equation $t = \mathrm{T}(t)$.

A subtler problem with using terms in my program analysis is that the symbol names in type terms are not useful. The type inference algorithm only uses symbol names to detect type errors; e.g. when the program requires a value to have both a function type and a scalar type. However, Ajax never rejects a program and does not need to detect these "errors".

2.2. The SEMI constraint system

It is natural to represent a term as a tree, where the root of the tree is labeled with the term's N-arity symbol and has N edges, labeled 0 to N-1. Edge i leads to a subtree representing the subterm at argument position i. A recursive type can be represented as an infinite tree, the "infinite unfolding" of the recursive definition. Of course, in practice the representation must be finite, and it is natural to simply represent the recursive type as a cyclic graph. In other words "μt.T" is represented as a rooted graph in which t is associated with the root node and references to t in T are simply edges pointing back to the root.

Therefore I discard terms and work directly with the graph structure. The graph edges that connect a node to its arguments become "component constraints". For example, the function type t = "$a \to b$" becomes the two constraints $\{ t \rhd_{\mathrm{arg}} a, t \rhd_{\mathrm{result}} b \}$. With component constraints, recursive structures require no special machinery. Furthermore, we shall see that the properties of polymorphic recursion become easy to state, in a way that is close to the implementation.

The SEMI solver uses the following structures:

- V — the set of variables (which can be thought of as type variables).
- L — the set of component labels (e.g."arg", "result"). SEMI treats them as abstract entities and assigns no meaning to them.
- I — the set of instance labels; each instance label represents a program site at which a polymorphic value is being used. SEMI treats them as abstract entities and assigns no meaning to them.
- C — a set of constraints of the following kinds:
    - "$a = b$" — an equality constraint expressing the fact that the two variables a and b are to be considered identical.
    - "$a \rhd_c b$" — a component constraint expressing the fact that variable a's component with label c is variable b.
    - "$a \preccurlyeq_i b$" — an instance constraint expressing the fact that variable a's instance i is variable b.

Later in this document I introduce additional constraints to support certain optimizations.

If the constraint $a \preccurlyeq_i b$ is present in a set, then I write "b is an instance of a" and "a is a source of b". The set should be clear from context. Likewise, if "$a \triangleright_c b$" is in a set, then I write "b is a component of a" and "a is a parent of b".

I also use transitive versions of the instance and component relationships:

- $a \underline{\preccurlyeq}_{\mathrm{C}} b \overset{\text{def}}{\equiv} \exists i_1, \ldots, i_n, t_1, \ldots, t_{n\text{-}1}.\ \{\ a \preccurlyeq_{i1} t_1, \ldots, t_{n\text{-}1} \preccurlyeq_{in} b\ \} \subseteq \mathrm{C}$
- $a \trianglerighteq_{\mathrm{C}} b \overset{\text{def}}{\equiv} \exists c_1, \ldots, c_n, t_1, \ldots, t_{n\text{-}1}.\ \{\ a \triangleright_{c1} t_1, \ldots, t_{n\text{-}1} \triangleright_{cn} b\ \} \subseteq \mathrm{C}$

2.3. Closure

I now define what it means for a set of constraints C to be closed. Closure can be thought of as a guarantee that the set is a consistent description of types and their relationships.

- **Equality closure rules**
  Equality has the usual properties of reflexivity, symmetry, transitivity, and substitutional equivalence. (Reflexivity is implicit; constraints of the form "$u = u$" are not required to be part of the constraint set.)
  $\{\ t = u\ \} \subseteq \mathrm{C} \Rightarrow \{\ u = t\ \} \subseteq \mathrm{C}$
  $\{\ t = u, u = v\ \} \subseteq \mathrm{C} \Rightarrow \{\ t = v\ \} \subseteq \mathrm{C}$
  $\{\ t = u, t \triangleright_c v\ \} \subseteq \mathrm{C} \Rightarrow \{\ u \triangleright_c v\ \} \subseteq \mathrm{C}$
  $\{\ t = u, v \triangleright_c t\ \} \subseteq \mathrm{C} \Rightarrow \{\ v \triangleright_c u\ \} \subseteq \mathrm{C}$
  $\{\ t = u, t \preccurlyeq_i v\ \} \subseteq \mathrm{C} \Rightarrow \{\ u \preccurlyeq_i v\ \} \subseteq \mathrm{C}$
  $\{\ t = u, v \preccurlyeq_i t\ \} \subseteq \mathrm{C} \Rightarrow \{\ u \preccurlyeq_i t\ \} \subseteq \mathrm{C}$
- **Component consistency rule**
  A given component of a variable has only one value. This and the previous rules together perform a kind of unification, effectively a monomorphic type inference.
  $\{\ t \triangleright_c u, t \triangleright_c v\ \} \subseteq \mathrm{C} \Rightarrow \{\ u = v\ \} \subseteq \mathrm{C}$
- **Instance consistency rule**
  A particular instantiation maps every occurrence of a given variable to the same new variable.
  $\{\ t \preccurlyeq_i u, t \preccurlyeq_i v\ \} \subseteq \mathrm{C} \Rightarrow \{\ u = v\ \} \subseteq \mathrm{C}$
- **Component propagation rule**
  Components propagate through instances.
  $\{\ t \preccurlyeq_i u, t \triangleright_c v\ \} \subseteq \mathrm{C} \Rightarrow \exists w.\ \{\ u \triangleright_c w\ \} \subseteq \mathrm{C}$
- **Instance propagation rule**
  Instances propagate down matching components
  $\{\ t \preccurlyeq_i u, t \triangleright_c v, u \triangleright_c w\ \} \subseteq \mathrm{C} \Rightarrow \{\ v \preccurlyeq_i w\ \} \subseteq \mathrm{C}$

2.4. Solver specification

Given an initial constraint set $\mathrm{C_I}$, the job of the solver is simply to find a closed set C containing $\mathrm{C_I}$.

C_I represents constraints induced by the program under analysis. C represents an extension of those constraints into a complete and consistent description of the "types" in the program.

Note that such a C always exists. For example, given C_I, add constraints making all variables equal and making all component and instance relationships hold between all variables. (This is finite because only the variables, component labels and instance labels that occur in C_I need be considered.) Effectively this merges everything into one giant type. In practice this result would not be useful, and it is preferable to retain distinctions between types whenever possible, but this example illustrates that implementations of the specification can trade off accuracy for performance.

2.5. Decidability and performance

It has been shown that the general problem of finding a principal type is undecidable in Henglein's type inference system. However, in practice all examples seem tractable. In fact, the algorithm is reported [H93] to be quite efficient at inferring types for functional programs.

To a large extent, these results carry over into my domain. In my work on the solver, which is similar in structure to Henglein's algorithm, nonterminating cases have always been traced back to errors in the solver implementation. It is also relatively efficient, allowing for the fact that Java programs induce constraint sets that are considerably more complex than the constraints induced by most functional programs. However, it is still true that my algorithm has no guarantee of termination. Clearly there are worst cases that give arbitrarily poor performance; efficiency depends on the characteristics of "average case" programs.

In fact, as noted above, there is no unique solution to my problem of finding a closed set, and such a set can always be found; my problem is trivially decidable. The goal is simply to obtain a "good" solution efficiently[1]. ("Good" generally means that the solver refrains from making variables equal, when that is possible.) It turns out that "optimal" solutions (which would correspond to principal types) do not actually exist in my domain, due to the combination of unrestricted recursive types and polymorphic recursion (see Appendix A for a proof).

Therefore, because termination of my algorithm is not guaranteed and optimal solutions do not exist, I measure performance and precision empirically. That is why I present no worst case bounds on the behavior of my algorithms.

2.6. Refined specification

The SEMI analysis engine extracts a "value-point relation" $\leftrightarrow_{P,C}$ from the closed set C. This relation is the only function of C that is used. Therefore I relax the specification of the solver to allow it to produce any set C' that gives the same value-point relation as the

[1] For this reason, I could guarantee termination by timing out and falling back to an algorithm that is guaranteed to terminate, but I have not found this to be necessary.

closed set C. I will call such a set "quasi-closed". This relaxation enables many optimizations.

$\leftrightarrow_{P,C}$ is defined as follows:

$B_1 \leftrightarrow_{P,C} B_2 \overset{\text{def}}{=} \exists x.\ M(B_1) \precsim_C x \wedge M(B_2) \precsim_C x$

M is a map from bytecode expressions (e.g. a variable in the bytecode language) to constraint variables.

This definition means that SEMI checks to see if two expressions are related by first converting them to constraint variables using the map M, then trying to find some variable x that is an instance of both variables (possibly transitively). This can be done by traversing the graph of $\preccurlyeq$ constraints.

The range of M is required to always be contained in the variables occurring in C_I. When querying on expressions that do not occur in the program text, the system may add constraints to C_I to associate constraint variables with these expressions. For example, if R is a local variable, then there will be some associated M(R) giving the constraint variable for R. Suppose R has a field P that is never accessed by the program, but there is a query on the expression "R.P". The constraints induced by the program may not associate a variable with "R.P". The system will need to add the constraint "$M(R) \rhd_P t$" to the initial set so that M(R.P) can return t.

From these definitions, it follows that C' is quasi-closed if there exists a C such that

- C is closed
- C contains C_I
- $\forall t, u \in \mathrm{Vars}(C_I).\ (\exists x.\ t \precsim_{C'} x \wedge u \precsim_{C'} x) \Leftrightarrow (\exists x.\ t \precsim_C x \wedge u \precsim_C x)$

2.7. Incrementality

The SEMI solver is incremental, in the sense that one can add to the initial constraint set C_I at any time and the (quasi) closed set C will correspondingly expand. This is not difficult to implement and is not discussed further in this document.

2.8. The graph representation of the value-point relation

I have described how to treat the computed value-point relation as a graph in order to make complex queries execute more efficiently [O99]. It is not hard to transform the graph of $\preccurlyeq$ constraints into the required form, but I will not detail the transformation here. In this document I will only describe how to extend an initial constraint set to a quasi-closed state.

3. Constraints for Java bytecode programs

Although the details of the extraction are beyond the scope of this report, I give examples to show how the constraints are used.

	Bytecode	Induced Initial Constraints	
	`class X {`	$T_f \preccurlyeq_{\text{f-in-X}} t_{X,f}$	$C_X \triangleright_f t_{X,f}$
	`f(a) {`	$T_f \triangleright_{\text{arg-0}} L_{\text{this-param,f}}$ $T_f \triangleright_{\text{result}} R_f$	$T_f \triangleright_{\text{arg-1}} L_{\text{a-param,f}}$
0:	`aload this;`		
1:	`areturn;`	$R_f = L_{\text{this-param,f}}$	
	`}`		
	`static g(c, d) {`	$T_g \triangleright_{\text{arg-0}} L_{\text{c-param,g}}$ $T_g \triangleright_{\text{result}} R_g$	$T_g \triangleright_{\text{arg-1}} L_{\text{d-param,g}}$
0:	`aload c`		
1:	`aload d`		
2:	`invokevirtual f;`	$L_{\text{c-param,g}} \triangleright_f u$ $u \triangleright_{\text{arg-1}} L_{\text{d-param,g}}$	$u \triangleright_{\text{arg-0}} L_{\text{c-param,g}}$
3:	`areturn;`	$u \triangleright_{\text{result}} R_g$	
	`}`		
	`static main(b) {`	$T_{\text{main}} \triangleright_{\text{arg-0}} L_{\text{b-param,main}}$	$T_{\text{main}} \triangleright_{\text{result}} R_{\text{main}}$
0:	`new X;`	$C_X \preccurlyeq_{\text{new-at-0-in-main}} L_{\text{tmp-1,main}}$	
1:	`aload b;`		
2:	`invokestatic g;`	$T_g \preccurlyeq_{\text{main-2}} t$ $t \triangleright_{\text{arg-1}} L_{\text{b-param,main}}$	$t \triangleright_{\text{arg-0}} L_{\text{tmp-1,main}}$
3:	`areturn;`	$t \triangleright_{\text{result}} R_{\text{main}}$	
	`}`		
	`}`		

Figure 1: Constraints for a bytecode program

3.1. Example

See Figure 1. Bytecode equivalent to the code in Figure 1 would be generated by the following Java code:

```
class X {
           X f(X a)       { return this; }
    static X g(X c, X d)  { return c.f(d); }
    static X main(X b)    { return g(new X(), b); }
}
```

For this program, one might ask "can `main`'s result equal the new X object it creates?" We shall see how this question is answered by extracting the initial constraints (shown in Figure 1) and finding a closed form.

3.2. The initial constraints

Consider the code and initial constraints for function `f`. T_f is the constraint variable associated with `f`. `f`'s parameters and results are also have constraint variables associated with them: $L_{this\text{-}param,f}$, $L_{a\text{-}param,f}$ and R_f. Component constraints record the relationship between these constraint variables and T_f. Returning "this" from `f` induces the equality constraint $R_f = L_{this\text{-}param,f}$.

The class X is associated with a constraint variable C_X; this can be thought of as the inferred "type" for a prototypical object of class X. C_X has a component for each nonstatic method of X. In this case, there is just one nonstatic method, `f`. Thus, C_X has an `f`-component, $t_{X,f}$. $t_{X,f}$ is an instance of T_f, because multiple classes may inherit `f` and we should keep these different uses of `f` separate, so that placing a constraint on one instance doesn't unnecessarily constrain the other instances.

A constraint variable associated with an object reference not only has components for the virtual methods of that object, it can also have components for the fields of that object. The field components are used by "putfield" and "getfield" instructions to access the inferred "type" of the field. The constraint variable for the class itself (e.g. C_X) has no field components, because nothing is known about the inferred types of fields until code is found to be operating on actual object instances.

Virtual method components are used when a virtual method is called, such as in the code for `g`. `g` has a reference to an X in its `c` parameter, associated with variable $L_{c\text{-}param,g}$. To call the virtual method `f` on `c`, the `f`-component of $L_{c\text{-}param,g}$ is extracted (into variable "u") and its result and parameter variables are bound to variables for the actual parameters and results. Note that "u" itself is used, not an instance of "u". Using an instance of "u" would make the analysis unsound; the intuitive reason is that only constant values can be treated polymorphically.

Static method calls can use a fresh instance of the called method's constraint variable, because the callee is a constant. In `main`, the constraint variable "t" is a fresh instance of the constraint variable for the callee, `g`.

When a new object is created, the variable corresponding to the new object ($L_{tmp\text{-}1,main}$ in this example) is made a fresh instance of the class's associated constraint variable (C_X). Thus the components for the virtual methods of X will propagate to the instance.

3.3. Finding a closed form

Closing the constraint set generates the following additional constraints (among others):

The equality constraints within *f* give

$T_f \rhd_{result} L_{this\text{-}param,f}$

We propagate components of T_f to $T_{x,f}$, getting

$t_{X,f} \triangleright_{\text{arg-0}} v \qquad t_{X,f} \triangleright_{\text{result}} v$

(where $L_{\text{this-param},f} \preccurlyeq_{\text{f-in-X}} v$)

Now we propagate $T_{x,f}$ and its components to the instance of C_X in main:

$L_{\text{tmp-1,main}} \triangleright_f t'_{X,f} \qquad t'_{X,f} \triangleright_{\text{arg-0}} v' \qquad t'_{X,f} \triangleright_{\text{result}} v'$

(where $v \preccurlyeq_{\text{new-at-0-in-main}} v'$)

In other words, we know in *main* that the object's *f* method aliases its first parameter and result. Now we need to work on *g*. *g*'s equality constraints give us

$L_{\text{c-param},g} \triangleright_f u \qquad u \triangleright_{\text{arg-0}} L_{\text{c-param},g} \qquad u \triangleright_{\text{result}} R_g$

So inside *g*, we know that we pass *c* into *c*'s *f* method, and the result of that method is returned from *g*. We don't assume anything else about *f* here.

We propagate *g*'s constraints to *main*:

$L_{\text{c-param},g} \preccurlyeq_{\text{main-2}} L_{\text{tmp-1,main}} \qquad R_g \preccurlyeq_{\text{main-2}} R_{\text{main}}$

From here we get

$L_{\text{tmp-1,main}} \triangleright_f u' \qquad u' \triangleright_{\text{arg-0}} L_{\text{tmp-1,main}} \qquad u' \triangleright_{\text{result}} R_{\text{main}}$

(where $u \preccurlyeq_{\text{main-2}} u'$).

Now $L_{\text{tmp-1,main}} \triangleright_f u'$ and $L_{\text{tmp-1,main}} \triangleright_f t'_{X,f}$ require us to set $u' = t'_{X,f}$. In other words, we have "discovered" the implementation of *f* that *g* uses.

From the arg-0 components of u' and $t'_{X,f}$, we get

$v' = L_{\text{tmp-1,main}} \qquad v' = R_{\text{main}}$

Thus

$L_{\text{tmp-1,main}} = R_{\text{main}}$

The conclusion is that the result of `main` may be the new X it creates; at least, this analysis cannot prove otherwise.

4. The basic algorithm

I will describe a series of algorithms leading up to the full SEMI algorithm, each more sophisticated than the last. All the algorithms start with the initial constraint set C_I and add constraints to the set until it is closed (or quasi-closed). The basic algorithm presented in this section corresponds to Henglein's type inference procedure [H93].

4.1. Equality elimination

Like every algorithm of this kind, my solver uses a representation of the constraint set that avoids explicit equality constraints. Whenever a constraint of the form "$a = b$" is encountered or produced, it is discarded, and the solver substitutes *b* for *a* (or *a* for *b*) in all the other constraints. This can be implemented efficiently by treating each variable as an equivalence class and employing the union-find algorithm to merge equivalence classes.

4.2. Functional representation of components and instances

The component consistency rule guarantees that for a given variable t and component label c, there is at most one v such that $t \rhd_c v$ (after taking into account equivalencies). Thus the component constraints can be represented as a curried partial function $F_{\rhd}$: $V \rightarrow L \mapsto V$.

Likewise the instance constraints can be represented as $F_{\preccurlyeq} : V \rightarrow I \mapsto V$.

In the implementation, each variable v has two hash tables associated with it, one representing $F_{\rhd}(v)$ and the other $F_{\preccurlyeq}(v)$.

When a variable v is substituted for u because u and v have been set equal, u's $L \mapsto V$ component map is merged into v's $L \mapsto V$ component map. The tricky part of this process is that, for each l in the intersection of their domains, the variable $F_{\rhd}(u)(l)$ is made equal to the variable $F_{\rhd}(v)(l)$; thus, the merge procedure can invoke itself recursively. The procedure corresponds to unification of terms.

The algorithm also has to merge u's $I \mapsto V$ instance map into v's $I \mapsto V$ instance map. This is similar to the case of the component maps, and can also result in recursive merge calls.

4.3. Component propagation

The above normalization procedures ensure that the constraint set is always closed under all rules, except for the component and instance propagation rules. Naturally, this is where things start to get interesting.

We treat the rules as production rules:

- **Component propagation**
  Upon detecting $\{ t \preccurlyeq_i u, t \rhd_c v \} \subseteq C$ for some t, u, v, i and c, add a new variable w and constraint $u \rhd_c w$ (unless there is already a w such that $u \rhd_c w$).
- **Instance propagation**
  Upon detecting $\{ t \preccurlyeq_i u, t \rhd_c v, u \rhd_c w \} \subseteq C$ for some t, u, v, w, i and c, add constraint $v \preccurlyeq_i w$ (if not already present).

This is implemented using a worklist. The algorithm maintains a list of "dirty" component constraints (e.g. "$t \rhd_c v$") that must be checked by the component propagation rule. All component constraints in C_I start off in the dirty list. Whenever a new component constraint is added, it is added to the dirty list. Whenever a variable t is substituted for another variable w, all the components of t that do not already appear in w are made dirty, and likewise all the components of w that do not already appear in t are made dirty; formally:

$$\{ t \rhd_c v \mid \{ t \rhd_c v \} \subseteq C \wedge (\neg \exists u.\ \{ w \rhd_c u \} \subseteq C) \} \cup$$
$$\{ w \rhd_c v \mid \{ w \rhd_c v \} \subseteq C \wedge (\neg \exists u.\ \{ t \rhd_c u \} \subseteq C) \}$$

Also, whenever an instance constraint $t \preccurlyeq_i u$ is added, all the components of t and u are made dirty.

During each iteration of the solver, it pulls one dirty component constraint $t \triangleright_c v$ from the dirty list. Then for each u and i such that $\{ t \preccurlyeq_i u \} \subseteq C$, the two production rules are checked. Also, for each u and i such that $\{ u \preccurlyeq_i t \} \subseteq C$, the second production rule is checked, swapping u with t and v with w. Note that in checking the second rule, since u and c are known, there can be at most one applicable w.

Iteration continues until the worklist of dirty component constraints is empty. Upon termination, the constraint set is closed.

When an equality constraint is processed by applying a substitution to the entire constraint set, the same substitution is applied to the elements of the worklist.

4.4. Saving time by recording additional dirtyness information

For some variables t there may be many u such that $t \preccurlyeq_i u$ or $u \preccurlyeq_i t$. When a dirty component $t \triangleright_c v$ is being processed, it can be slow to scan all the instances u such that $t \preccurlyeq_i u$ and all the sources u such that $u \preccurlyeq_i t$. Therefore for each dirty component $t \triangleright_c v$, we maintain a list of all the $t \preccurlyeq_i u$ and $u \preccurlyeq_i t$ that need to be inspected in conjunction with that component constraint. For every situation in which a component constraint may become dirty, there is an associated set of instance and source constraints that will need to be inspected. If a component constraint is already dirty, its associated sets can still be extended with additional instance and source constriants to be inspected.

- When a new component constraint $t \triangleright_c v$ is added, all constraints of the form $t \preccurlyeq_i u$ and $u \preccurlyeq_i t$ need to be inspected.
- When the variable t is substituted for variable w, then for each $t \triangleright_c v$ such that $\{ t \triangleright_c v \} \subseteq C \wedge (\neg\exists u.\ \{ w \triangleright_c u \} \subseteq C)$, all constraints of the form $w \preccurlyeq_i u$ and $u \preccurlyeq_i w$ need to be inspected in conjunction with $t \triangleright_c v$. Likewise, for each $w \triangleright_c v$ such that $\{ w \triangleright_c v \} \subseteq C \wedge (\neg\exists u.\ \{ t \triangleright_c u \} \subseteq C)$, all constraints of the form $t \preccurlyeq_i u$ and $u \preccurlyeq_i t$ need to be inspected in conjunction with $w \triangleright_c v$.
- Whenever an instance constraint $t \preccurlyeq_i u$ is added, then for each $t \triangleright_c v$ in C, the instance constraint $t \preccurlyeq_i u$ must be inspected in conjunction with $t \triangleright_c v$. Also, for each $u \triangleright_c v$ in C, the source constraint $t \preccurlyeq_i u$ must be inspected in conjunction with $u \triangleright_c v$.

This additional bookkeeping greatly improved runtime, while adding a small amount of space overhead (bounded by a small constant factor in the worst case, but still significant).

4.5. Nontermination problems

Unfortunately, it is easy to construct constraint sets for which this algorithm does not terminate. Furthermore, these sets do arise in practice.

For example, consider the set $\{ T_f \triangleright_{result} T_r,\ T_f \preccurlyeq_i T_r \}$. This could arise from an analysis of the following program:

```
f() { return f; }
```

`f`'s result is an instance of `f`. (This is a contrived example. Real examples in Java are more complicated, e.g. when a method M returns a reference to a new object which contains M.)

Suppose we apply the above algorithm to this constraint set:

- Apply component propagation to $\{ T_f \rhd_{\text{result}} T_r, T_f \preccurlyeq_i T_r \}$:
  add T_1 and constraint $\{ T_r \rhd_{\text{result}} T_1 \}$
- Apply instance propagation to $\{ T_f \rhd_{\text{result}} T_r, T_f \preccurlyeq_i T_r, T_r \rhd_{\text{result}} T_1 \}$:
  add constraint $\{ T_r \preccurlyeq_i T_1 \}$
- Apply component propagation to $\{ T_r \rhd_{\text{result}} T_1, T_r \preccurlyeq_i T_1 \}$:
  add T_2 and constraint $\{ T_1 \rhd_{\text{result}} T_2 \}$
- Apply instance propagation to $\{ T_r \rhd_{\text{result}} T_1, T_r \preccurlyeq_i T_1, T_1 \rhd_{\text{result}} T_2 \}$:
  add constraint $\{ T_1 \preccurlyeq_i T_2 \}$
- …

4.6. The extended occurs check

In type inference, the type of `f` would be an infinite term:

void → (void → (void → …))

This recursive type is not valid in Henglein's scheme; therefore his algorithm detects this situation and reports failure. He calls this detection the "extended occurs check". (It is analogous to the occurs check performed during term unification.) In terms of my formalism, the extended occurs check fires whenever, for some sets of variables t_i and u_i:

$$\{ t_1 \preccurlyeq_{i1} u_1, \ldots, u_{n\text{-}1} \preccurlyeq_{in} u_n, t_1 \rhd_{comp1} t_2, \ldots, t_m \rhd_{compn} u_n \} \subseteq C$$

This means that the extended occurs check is applicable whenever we have a variable t_1 that has a transitive instance u_n that is also transitively a component of t_1.

When the extended occurs check fires in SEMI, the solver simply forms a recursive type by setting $t_1 = u_n$, and continues. In the example, the extended occurs check sets $T_r = T_f$, halting the expansion.

Note that adding this equality forces variables to be equal that do not necessarily need to be equal according to the initial constraints. This is why SEMI does not compute a most general (i.e. principal) solution. The demonstration of non-existence of principal types in Appendix A is based on a similar example.

The implementation of the SEMI solver performs an extended occurs check whenever the instance propagation rule adds a new instance constraint $t \preccurlyeq_i u$ to C. It sets $u_{n\text{-}1} = t$, $u_n = u$, and $i_n = i$, and then searches the component and instance graphs for a variable t_1 satisfying the check. Any such variables found are bound to u. The search proceeds by first scanning the instance graph backwards, finding all candidate t_1s that are transitive sources of t (including t itself), and for each candidate, scanning its components transitively looking for u.

This check could easily be changed from worst case $O(N^2)$ time, where N is the number of variables, to O(N) time, simply by finding all transitive sources of t first, then scanning all of *t*'s transitive parents (variables that have *t* as a transitive component). In practice, however, the average numbers of transitive instances, sources, components or parents that a variable has are all very large, and a check that is linear time in any of these quantities is prohibitively expensive (since the extended occurs check is performed frequently). Therefore a more complex approach is required (described below), which builds on the basic algorithm above. It turns out that in the presence of those optimizations, the worst case O(N^2) version performs a lot better.

5. Optimizing the occurs check: clusters

The naïve approach to performing the extended occurs check can be sped up enormously by exploiting the structure of the constraints induced by a Java program (or any program that has layers in its architecture, i.e. almost all programs).

5.1. Constraint structure

SEMI generates instance constraints from a Java program in the following situations:

- A method body M_1 makes a "static" call to another method M_2 (M_1 depends on M_2)
- A method body M_1 creates a new object of a class C (M_1 depends on C)
- A method body M_1 is installed in the dynamic dispatch table of a class C (C depends on M_1)

Due to the layered structure of most programs, the graph of dependencies is "mostly" acyclic. However, the JDK class library itself contains a number of surprisingly complex cycles, so it is important to be able to handle cycles well.

5.2. Clusters

Normally (i.e., in the absence of a recursive chain of dependencies), the variables associated with parameters, local variables, results, and intermediate values within a given method, and their components, are related only by component constraints. Instance constraints (and **only** instance constraints) relate these variables to variables associated with other methods. Similarly, in a class there are variables associated with the method slots, and a variable for the prototype object of the class, which are related to each other by component constraints only. Instance constraints relate these variables to variables in the methods that create objects of the class, and to variables in the method bodies used by the class.

The SEMI solver explicitly captures this structure. The variables are partitioned into abstract *clusters*; the partition is written $R : V \to X$ (where X is the set of cluster labels). The only required property of R is that if $t \rhd_c u$ is a constraint, then $R(t) = R(u)$. In other words, all variables related by only component constraints are in the same cluster. Typically, Java programs give rise to a large number of small clusters (one cluster per method).

It is not strictly necessary to have R be the most refined partition possible, but that is easy to implement and gives the best results. That is, if t and u are not related by any chain of component constraints, ignoring direction, then $R(t) \neq R(u)$.

The implementation maintains the cluster map dynamically, taking account of variable merging, new constraints, etc.

5.3. Optimizing the extended occurs check using clusters

The cluster map is used to short-circuit the subroutine that computes "Is u a transitive component of t_1?". If $R(u) \neq R(t_1)$, then the result must be false. Since clusters are generally small and numerous, and following an instance constraint usually leads to another (different) cluster, $R(u) \neq R(t_1)$ almost always holds during the extended occurs check search.

5.4. Cluster levels

Unfortunately, even scanning all transitive sources of a variable and performing a constant-time check for each is too expensive, given the frequency with which extended occurs checks are performed.

SEMI resolves this problem by explicitly capturing the "mostly acyclic" structure of the inter-cluster instance graph. The instance constraints are projected onto the clusters; i.e. the clusters are assembled into a directed graph G such that for each $t \preccurlyeq_i u$, $(R(t), R(u))$ is an edge in G. Then the graph is partitioned into strongly connected components, called *cluster levels*. This partition is written $S : X \to Z$, where Z is the set of cluster level labels. By definition, G projected onto cluster levels is acyclic (excluding self-loops). The fact that G itself is "mostly acyclic" means that most cluster levels contain just one cluster.

The implementation maintains the cluster levels dynamically, as the underlying constraint system changes. I have developed an efficient implementation, but it is tricky, because detecting cycles can be expensive. I found it helpful to delay cycle detection (in response to changes in the constraint system) until the cluster levels are required to be in a consistent (acyclic) state (e.g., until the next extended occurs check). My system maintains a "dirty" bit for each cluster level, indicating that it may be part of a cycle of cluster levels because of the addition of new instance constraints incident to the cluster level. When the acyclic state is required, the algorithm performs a worst-case linear time traversal of the cluster level graph — a depth-first search backwards along the instance edges, starting from the dirty cluster levels. Any cycles found are recorded. Finally, the cluster levels in each cycle are merged. It requires care to make sure that **all** cycles are detected, since the straightforward depth-first search algorithm is only guaranteed to find one cycle (assuming a cycle exists).

In my system, the cost of maintaining the cluster levels is usually negligible and never the performance bottleneck.

5.5. Optimizing the extended occurs check using cluster levels

The cluster level map is used to optimize the subroutine that scans the source graph for all candidate t_1s that are transitive sources of t.

The extended occurs check subroutine receives t and u where u is an instance of t. Therefore every candidate t_1 has u as a transitive instance. Now suppose $S(R(u)) \neq S(R(t_1))$. There must be a path from $S(R(t_1))$ to $S(R(u))$ in the instance graph projected onto the cluster levels, because there is a path from t_1 to u in the instance graph. Because the cluster level instance graph is acyclic, there cannot be a path from $S(R(u))$ to $S(R(t_1))$. Therefore, for all transitive sources s of t_1, $S(R(s)) \neq S(R(u))$, because otherwise we would have an instance path from $S(R(s)) = S(R(u))$ to $S(R(t_1))$.

Therefore, whenever the extended occurs check subroutine detects $S(R(u)) \neq S(R(t_1))$, t_1's sources need not be searched. In practice this prunes the search tremendously. In particular, if $S(R(u)) \neq S(R(t))$ then $R(u) \neq R(t)$ follows, and neither t nor its sources need be checked; the entire check takes constant time.

In the special case in which there are no recursive dependencies in the original program, the instance graph projected onto clusters is acyclic, i.e. S is one-to-one. Then the extended occurs check always completes in constant time. In other words, this optimization ensures that the extended occurs check only incurs a cost when polymorphic recursion is actually being used.

5.6. Replacing the extended occurs check with a conservative approximation

In the case $S(R(u)) = S(R(t))$, instead of performing the rest of the extended occurs check, one could simply add the equality constraint "$u = t$". The new instance constraint $t \preccurlyeq_i u$ is reduced to a self-loop in the instance graph, which forestalls the nonterminating behavior that the extended occurs check is designed to prevent. This approach is similar to the Hindley-Milner algorithm, which (interpreted in this context) simply prohibits any polymorphism constraints within a cluster level. This behavior can lead to smaller constraint sets because of the "unnecessary" inequalities that are introduced, which improves performance but does yield a noticeable decrease in accuracy for some applications of the analysis.

5.7. Possible "lazy" extended occurs checking

I considered several schemes for reducing the frequency and amortized cost of extended occurs checks by postponing them and processing several at once in a single search. This seemed to be extremely difficult, partly due to the complexities of implementation, but also because once an extended occurs violation arises, the solver can repeatedly "unwind" the structure to generate a large number of new constraints. Thus it seems necessary to detect and correct the situation as soon as possible.

In any case, further optimization of the extended occurs check seems unprofitable at this time. The amount of time and space the solver spends on the task is currently negligible.

6. Scheduling the worklist using cluster levels

It turns out that the acyclic cluster level graph is useful for tasks other than optimizing the extended occurs check.

6.1. The scheduling problem

Components propagate from sources to instances, but not the other way around. Therefore as changes are made to constraints at the "bottom" of the instance graph, they tend to "bubble" up to instances. It improves performance to do as much work as possible at the bottom of the instance graph before making changes further up the graph, by reducing the number of times each component is visited or examined.

6.2. Using cluster levels

A cluster level *l* is "dirty" if there is a component constraint in the worklist of the form "$t \rhd_c u$", where S(R(t)) = *l*.

Whenever SEMI chooses a component constraint from the worklist, it chooses a constraint $t \rhd_c u$ where the cluster level S(R(t)) has no dirty cluster levels below it in the projected instance graph. Such a constraint is guaranteed to exist because the cluster level graph is acyclic.

Making this choice efficiently is tricky, but requires negligible time and space in my system. The dirty component constraints are stored on the worklist indexed by cluster levels; the problem reduces to finding an appropriate cluster level to work on. I record in each cluster level whether it is dirty. I also cache two facts in each cluster level: whether it is known that there are *some* dirty cluster levels below it on the projected instance graph, and whether it is known that there are *no* dirty cluster levels below it on the projected instance graph. In practice, this cache can be updated and invalidated efficiently in response to changes in dirty state and changes in the underlying constraint set.

The system keeps a list of dirty cluster levels, separated into two parts: the set of dirty cluster levels that are known to have no dirty cluster levels below them on the projected instance graph, and the rest. When a constraint is selected from the worklist, a cluster level is chosen from the former set and one of its dirty constraints is used. If the former set is empty, then a cluster level *l* is chosen from the latter set. Then the algorithm performs a depth-first search of the cluster level projected instance graph, backwards from *l*, from instances to sources. During this search, each visited cluster level is marked as either having dirty cluster levels below it, or not. The acyclic nature of the projected instance graph guarantees that after this procedure, at least one dirty cluster level will be found with no dirty cluster levels below it (unless there are no dirty cluster levels left, in which case the algorithm terminates).

7. Suppressing components: advertisements

7.1. Useless component propagation

Suppose F is a function in the program for which we infer a large "type", T_F. This means that T_F is the root of a large graph of component constraints. At every use of F (a direct call or the use of F to fill a slot in a method table), a new instance i of T_F is created, and a constraint $T_F \preccurlyeq_i t$ is added. The component propagation rule will effectively copy the transitive components of T_F (i.e., the component graph under T_F) to the instance. Often, however, much of this structure will not be used. For example, consider this Java code:

```
Foo x = bar();
println(x.kitty);
```

Given the code for `bar`, the analysis may work out some complex type structure for its return value, including information about the various methods and fields of `x`. All this information will be propagated to the caller, but only one field is used, and therefore the rest of the information is irrelevant.

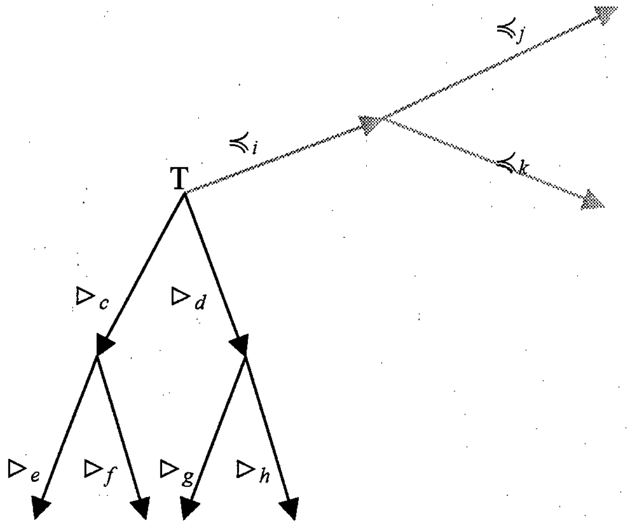
Furthermore, suppose `bar` is implemented as a wrapper:

```
Foo bar() { return baz(5); }
```

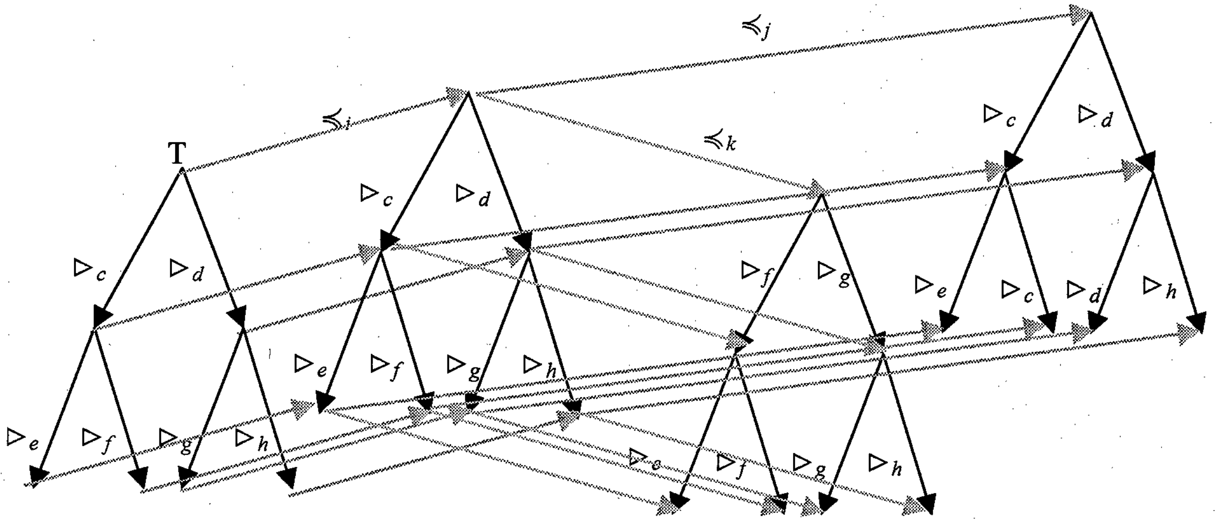
Such constructs are common, and defeat purely local schemes for suppressing useless structure.

7.2. Illustration

Consider the following constraint set Q. This diagram and the diagrams that follow represent graphs. Nodes correspond to variables. A constraint of the form $t \rhd_c u$ is displayed as a black edge from t's node to u's node labelled with $\rhd_c$. A constraint of the form $t \preccurlyeq_i u$ is displayed as a gray edge from t's node to u's node labelled with $\preccurlyeq_i$.



T represents the type of some compound object with an instance *i* and further instances *j* and *k*. Assume Q contains the initial constraint set, C_I. The basic algorithm extends Q to the following closed set C:



The basic algorithm reaches C by copying T's component tree to all the instances, and connecting the components with instance relationships.

7.3. Quasi-closure conditions

These new components are all unnecessary; Q is, in fact, quasi-closed. To see this, consider two variables in C_I, *u* and *v*. I must show that *u* and *v* are related in Q if and only if they are related in C. There are two cases:

- Suppose *u* and *v* are not related in C. Then $\neg\exists x.\ u \precsim_C x \wedge v \precsim_C x$. It follows that $\neg\exists x.\ u \precsim_Q x \wedge v \precsim_Q x$, since C is a superset of Q.

- Suppose u and v are related according to C. Then $\exists x.\ u \precsim_{\mathrm{C}} x \wedge v \precsim_{\mathrm{C}} x$. I show that $\exists p.\ u \precsim_{\mathrm{Q}} p \wedge v \precsim_{\mathrm{Q}} p$, by induction on the length of the shortest chain of instances justfiying $u \precsim_{\mathrm{C}} x$.
  Regardless of the length of the chain, if x occurs in Q, then $u \precsim_{\mathrm{Q}} x \wedge v \precsim_{\mathrm{Q}} x$, since the chains of instances justifying $u \precsim_{\mathrm{C}} x$ and $v \precsim_{\mathrm{C}} x$ are also in Q. (In other words, every instance constraint in C that holds between variables in Q is is already in Q.). Thus the induction hypothesis holds, setting $p = x$.
  If the length of the chain is zero, then $x = u$, hence x is in Q and the hypothesis holds. If x is not in Q, then it must be a child variable of one of the new component constraints. Each such variable has a unique predecessor P_x such that $\mathrm{P}_x \preccurlyeq x$. The chains $u \precsim_{\mathrm{C}} x$ and $v \precsim_{\mathrm{C}} x$ must have length at least one, since x is not in Q and therefore does not equal u or v. Therefore the last link of each chain must be $\mathrm{P}_x \preccurlyeq x$. Therefore, it also holds that $u \precsim_{\mathrm{C}} \mathrm{P}_x \wedge v \precsim_{\mathrm{C}} \mathrm{P}_x$. By the induction hypothesis, $\exists p.\ u \precsim_{\mathrm{Q}} p \wedge v \precsim_{\mathrm{Q}} p$.

This argument can be generalized. A general set Q is quasi-closed over $\mathrm{C_I}$ if:

1. Equalities have been eliminated from Q, and it is closed under the instance and component consistency rules (guaranteed by my representation).
2. Q contains $\mathrm{C_I}$.
3. Q is closed under the instance propagation rule.
4. For all t, u, v, c, x, y, if $t \precsim_{\mathrm{Q}} u \wedge u \precsim_{\mathrm{Q}} v \wedge \{ t \rhd_c x, v \rhd_c y \} \subseteq \mathrm{Q}$, then there is a w such that $\{ u \rhd_c w \} \subseteq \mathrm{Q}$.
5. For all t, u, c, v, if $t \precsim_{\mathrm{Q}} u \wedge \{ t \rhd_c v \} \subseteq \mathrm{Q}$ but $\{ u \rhd_c w \} \not\subseteq \mathrm{Q}$ for any w, then the set $\{ x \mid \exists j, w, y.\ y \precsim_{\mathrm{Q}} u \wedge \{ x \preccurlyeq_j y, x \rhd_c w \} \subseteq \mathrm{Q} \wedge (\forall z.\ \{ y \rhd_c z \} \not\subseteq \mathrm{Q}) \} = \{ t \}$.

Conditions 1 and 2 are fundamental. Conditions 3 and 4 are required to justify the "x in Q" part of the proof; they require Q to be closed except possibly for some unexpanded instances of compound structures. Condition 5 is required to justify the "x not in Q" part of the proof; it ensures that if a component c is not propagated to u, then there is a unique instance-chain predecessor that has a real component that we can fall back to. I have not worked out the details of the argument yet, but this is what the system does.

7.4. Advertisements

The system reaches this state by propagating components lazily. When the component propagation rule fires, it actually propagates an *advertisement*, representing the possibility of a component being present in the instance. An advertisement is a pair: the parent variable, v, and a component label, c, written $v \succcurlyeq_c$. These advertisements are propagated along the instance graph using two rules:

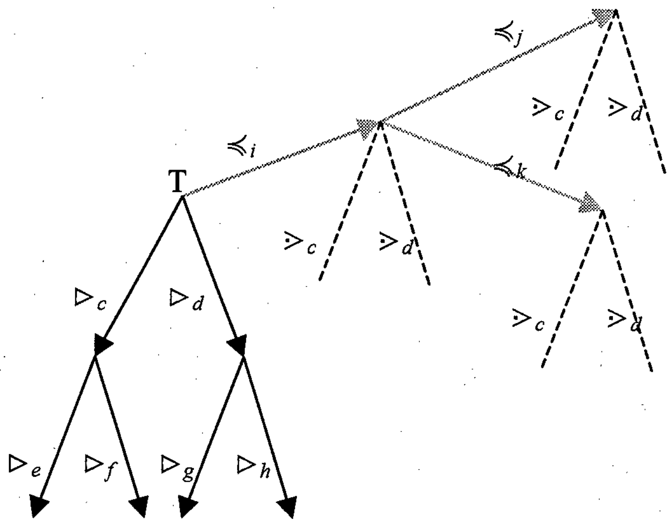
- **Advertisement propagation from component**
  Upon detecting $\{ t \preccurlyeq_i u, t \rhd_c v \} \subseteq \mathrm{C}$ for some t, u, v, i and c, add $u \succcurlyeq_c$.
- **Advertisement propagation from advertisement**
  Upon detecting $\{ t \preccurlyeq_i u, t \succcurlyeq_c \} \subseteq \mathrm{C}$ for some t, u, i and c, add $u \succcurlyeq_c$.

If a variable t already has a component c, then it does not need an advertisement for the same component.

- **Redundant advertisement suppression**
  Upon detecting $\{ t \gtrdot_c, t \rhd_c v \} \subseteq C$ for some t, v and c, delete $t \gtrdot_c$.

The component propagation rule from the basic algorithm is not used, but the instance propagation rule still is. Thus the algorithm will terminate with a set satisfying at least quasi-closure conditions 1, 2 and 3.

7.5. Example



Instead of copying T's entire component tree, we have added advertisements for T's immediate components.

7.6. Ensuring quasi-closure: fill-in

To satisfy quasi-closure condition 4, the algorithm "fills in" an advertisement that has a real component above it in the instance graph:

- **Advertisement fill-in**
  Upon detecting $\{ t \preccurlyeq_i u, t \gtrdot_c, u \rhd_c w \} \subseteq C$ for some t, u and w, add $t \rhd_c v$, where v is a fresh variable.

For example, consider the initial set:

$$\begin{tikzcd}
T \arrow[r, "\preccurlyeq_i"] \arrow[d, "\rhd_c"'] & {} \arrow[r, "\preccurlyeq_j"] & U \arrow[d, "\rhd_c"'] \\
T_c & & U_c
\end{tikzcd}$$

An advertisement will be added between T and U. The fill-in rule will ensure that the advertisement is replaced with a real component. The instance propagation rule will ensure that the instance chain from T_c to U_c is completed:

$$\begin{tikzcd}
T \arrow[r, "\preccurlyeq_i"] \arrow[d, "\rhd_c"'] & {} \arrow[r, "\preccurlyeq_j"] \arrow[d, "\rhd_c"'] & U \arrow[d, "\rhd_c"'] \\
T_c \arrow[r, "\preccurlyeq_i"] & {} \arrow[r, "\preccurlyeq_j"] & U_c
\end{tikzcd}$$

7.7. Ensuring quasi-closure: detecting conflicting sources

To satisfy quasi-closure condition 5, each advertisement is associated with an *advertisement source*, *s*, that records the variable the advertisement is derived from. The advertisement is written $t \gtrdot_c [s]$. Quasi-closure condition 5 becomes the "unique source condition":

If the advertisement $u \gtrdot_c [s]$ exists, then the set $\{ x \mid \exists j, w, y.\ \{ x \preccurlyeq_j y, y \preceq_C u, x \rhd_c w \} \subseteq C \wedge (\forall z.\ \text{“}y \rhd_c z\text{”} \notin C) \} = \{ s \}$.

The advertisement rules are extended:

- **Advertisement propagation from component**
  Upon detecting $\{ t \preccurlyeq_i u, t \rhd_c v \} \subseteq C$ for some *t*, *u*, *v*, *i* and *c*, add $u \gtrdot_c [t]$.
- **Advertisement propagation from advertisement**
  Upon detecting $\{ t \preccurlyeq_i u, t \gtrdot_c [s] \} \subseteq C$ for some *t*, *u*, *s*, *i* and *c*, add $u \gtrdot_c [s]$.
- **Redundant advertisement suppression**
  Upon detecting $\{ t \gtrdot_c [s], t \rhd_c v \} \subseteq C$ for some *t*, *v*, *s* and *c*, delete $t \gtrdot_c [s]$.
- **Advertisement fill-in**
  Upon detecting $\{ t \preccurlyeq_i u, t \gtrdot_c [s], u \rhd_c w \} \subseteq C$ for some *t*, *u*, *s* and *w*, add $t \rhd_c v$, where *v* is a fresh variable.

When a conflict arises — two advertisements for the same component show different sources — we collapse the advertisements and make a real component.

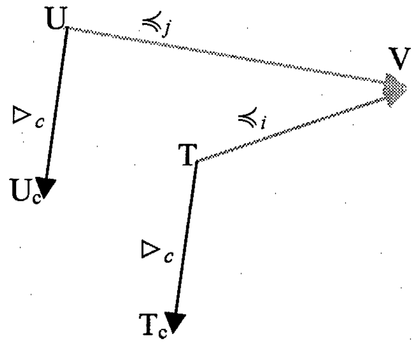
- **Conflicting advertisement detection**
  Upon detecting $\{ t \gtrdot_c [s], t \gtrdot_c [r] \} \subseteq C$ for some t, s, c and r, where $r \neq s$, create a new w and add $t \rhd_c w$.

This rule tests for the inequality of two variables. This could be tricky because variables can become equal during the run of the algorithm, but in fact it only means that conflicts may be detected that in the end may not be "true" conflicts. Since replacing an advertisement with a real component is always a conservative operation (possibly hurting performance, but never correctness), this is not a problem.

The conflicting advertisement rule guarantees that upon termination, the unique source condition is satisfied.

7.8. Simple example

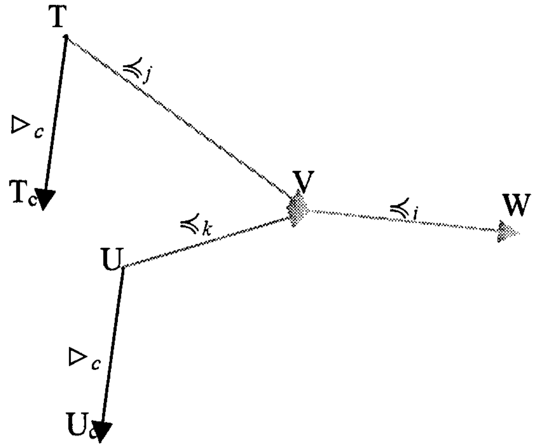
For example, consider this C_I:



The algorithm propagates advertisements from U and T to V, but since U ≠ T, the conflict detection rule fires and a real component is created for V. This is necessary to make the result quasi-closed.

7.9. Advertisement source updates

The conflicting advertisement detection rule alone is not satisfactory, however. Consider this example:



Suppose the algorithm propagates an advertisement from T to V and then W, and then propagates an advertisement from U to V. (This schedule might be chosen because of additional constraints not shown.) Now at V there are conflicting advertisements, with sources U and T. The algorithm creates a real component at V. Next it propagates an advertisement for that component to W. Now there are conflicting advertisements at W, with sources T and V, so a new component must also be created at W. This is suboptimal because W could simply have an advertisement with source V.

To avert such situations, it suffices to destroy the advertisements that could be affected by a new component; they will be regenerated with correct source information, if possible.

- **Advertisement source update**
  Upon detecting $\{ t \gtrdot_c [s], y \rhd_c z \} \subseteq C$ for some t, s, y, z and c, where $s \precsim_C y$, $y \precsim_C t$ and $s \neq y$, delete "$t \gtrdot_c [s]$".

7.10. Implementation

Advertisement constraints are easily added by treating them as a degenerate kind of component. Propagation and fill-in detection are implemented by allowing advertisements as well as components to be on the worklist. Conflicting advertisement detection is straightforward to implement and is done eagerly.

The advertisement source update is difficult to implement efficiently. The straightforward implementation can destroy and recreate many advertisements each time a component is added. My system uses an alternative representation for the source field of an advertisement. An advertisement for c at t records a "bottleneck variable" v such that every instance chain from the true source s to t passes through v. v may be s, or it may be some instance of s, in which case v also has an advertisement for c (and its own bottleneck variable, etc). The true source s for t can be found quickly; it is either v, or it is v's true source. When v is not s, components may be added along the path from s to v without having to update the information cached in the advertisement at t.

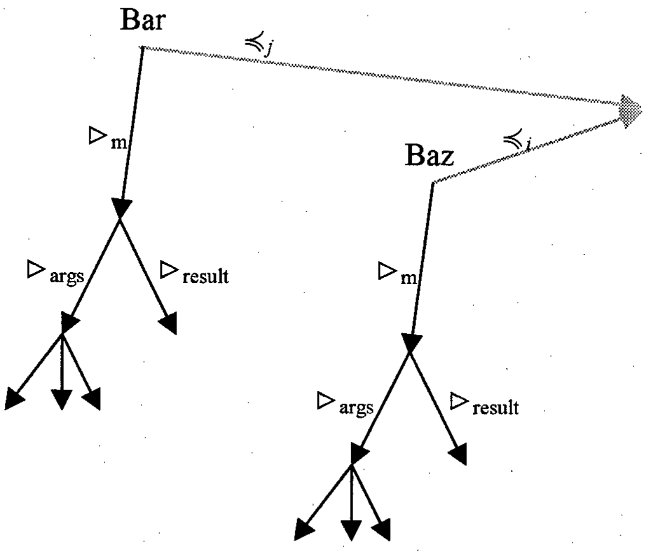
8. Suppressing components: modality

8.1. Example

Consider the following Java code:

```
Foo x = b ? new Bar() : new Baz();
println(x.kitty);
```

The advertisement algorithm does not perform well on this code. Suppose T_x is the constraint variable associated with `x`. For each dynamically dispatched method *m* defined in both classes `Bar` and `Baz`, T_x will get two advertisements for component *m*, one from `Bar` and one from `Baz`. If the method implementations are different, then the advertisements will have conflicting sources, so the structure of the method's inferred type will be expanded (forming the unification of the types of `Bar`'s *m* and `Baz`'s *m*). This can result in a large number of unnecessary constraints.



8.2. Approach

I annotate component constraints with *mode* information indicating how that component is used. A component constraint is written $t \rhd_c^- u$, $t \rhd_c^c u$, $t \rhd_c^d u$, or $t \rhd_c^{cd} u$. The superscript "c" means that the component is used in "constructor" mode. The superscript "d" means that the component is used in "destructor" mode. The superscript "-" means that the component is not used in any mode. "cd" means that the compnent is used in both modes.

The idea comes from the realm of functional languages. In that domain, component constraints are associated with the use of type constructors, such as the arrow type for functions. The type rules for these languages have two forms: one form that introduces a new occurrence of the constructor ("constructor mode"), e.g., the "lambda" rule for creating a new function, and another form that eliminates an occurrence of the constructor and uses the components ("destructor mode"), e.g., the "app" rule for applying a function. The intuition I rely on is that if a component is not used in both constructor and destructor modes, then no useful information is transmitted through it. For example, if a function type is introduced through the "lambda" rule but is never subject to the "app" rule, then it does not matter what its components are. Similarly, if there is an "app" with no corresponding "lambda" then the components do not matter. (In operational terms, the code performing the application must be dead.)

When SEMI gathers constraints from the original Java bytecode program, it adds mode annotations to the component constraints as follows:

- Installing a method implementation into a new object type adds a component constraint in constructor mode.
- Calling a virtual method in an object type adds a component constraint in destructor mode.
- Writing a field of an object type adds a component constraint in constructor mode.
- Reading a field of an object type adds a component constraint in destructor mode.
- Calling a method adds parameter and result component constraints to the method type in destructor mode.
- Declaring a method adds parameter and result component constraints to the method type in constructor mode.

This mode information changes the interface to the solver and its specification. The relevant change is in the definition of closure. The following parts of the definition of closure are altered:

- **Component propagation rule**
  Components propagate through instances, with nondecreasing modes:
  $\{\, t \preccurlyeq_i u, t \rhd_c{}^{m}\, v \,\} \subseteq \mathrm{C} \Rightarrow \exists w, m'.\ \{\, u \rhd_c{}^{m'}\, w \wedge m \subseteq m' \,\} \subseteq \mathrm{C}$

The benefit of modes is that we can safely inhibit some instance propagation.

- **Instance propagation rule**
  Instances propagate down matching components, if the modes match
  $\{\, t \preccurlyeq_i u, t \rhd_c{}^{m}\, v, u \rhd_c w \,\} \subseteq \mathrm{C} \wedge (\exists y, z.\ u \precsim_{\mathrm{C}} y \wedge \{\, y \rhd_c{}^{\mathrm{cd}}\, z \,\} \subseteq \mathrm{C}) \Rightarrow \{\, v \preccurlyeq_i w \,\} \subseteq \mathrm{C}$
  The instance constraint is only propagated to the component if there is some transitive instance of the component constraint that is used in both constructor and destructor mode. Otherwise the instance constraint need not be propagated.

8.3. Solver rules

The solver rules given in previous sections remain in force. Rules that match a component constraint match any mode annotation. Rules that add component constraints add constraints with the "no mode" annotation. We introduce a separate rule to propagate annotation information:

- **Mode propagation**
  Upon detecting $\{\, t \preccurlyeq_i u, t \rhd_c{}^{m}\, v, u \rhd_c{}^{m'}\, w \,\} \subseteq \mathrm{C}$ for some *t*, *u*, *v*, *i*, *c*, *w*, *m* and *m'*, replace "$u \rhd_c{}^{m'}\, w$" with "$u \rhd_c{}^{m \cup m'}\, w$".

- **Instance propagation**
  Upon detecting $\{\ t \preccurlyeq_i u,\ t \rhd_c^{m} v,\ u \rhd_c w\ \} \subseteq \mathrm{C}$ for some t, u, v, w, i, c, and m, if $\exists y, z.\ u \preceq_{\mathrm{C}} y \wedge y \rhd_c^{\mathrm{cd}} z$, then add constraint $v \preccurlyeq_i w$ (if not already present).

8.4. Implementation

These rules are not difficult to implement, and cost very little in time and space. Mode propagation takes place along with the other work on each dirty constraint from the worklist. The instance propagation check is performed very efficiently by tracking, for each $t \rhd_c v$, whether there is an instance of the component with the "cd" annotation; this "instance mode" information is propagated from instances to sources.

9. Globals

9.1. Encoding globals

It is straightforward to encode global variables ("static fields" in Java) in the constraint system presented. They can be treated as a single "globals" object with one field for each variable, which is passed into each function as a parameter. However, this is not very efficient because globals information must be propagated through each method type. It is much more efficient, and no less accurate, to have just one variable representing the globals object and one copy of the information for the global variables.

9.2. Characterization of constraints for globals

In terms of the constraints, a variable v in an initial set $\mathrm{C_I}$ is global if, for all closed sets C containing $\mathrm{C_I}$, $\exists g. \forall y.\ v \preceq_{\mathrm{C}} y \Rightarrow y \preceq_{\mathrm{C}} g$. This means that, for a variable corresponding to global data in some context, there is a corresponding "top level" variable representing the original global data that is passed into the "main" method and is used everywhere.

Suppose that variables t and u are related according to the quasi-closure check. Then $\exists x.\ t \preceq_{\mathrm{C}} x \wedge u \preceq_{\mathrm{C}} x$. If x corresponds to global data, then $\exists g. \forall y.\ x \preceq_{\mathrm{C}} y \Rightarrow y \preceq_{\mathrm{C}} g$. Therefore $\exists g.\ t \preceq_{\mathrm{C}} g \wedge u \preceq_{\mathrm{C}} g \wedge g$ is a top-level global. Thus, for quasi-closure, we do not need any copies of global variable information other than the top-level copy.

9.3. Implementation

SEMI gives hints to the solver that a variable corresponds to global data. The solver marks such variables as global and treats global variables specially:

- If $t \rhd_c v$ and t is global then v is made global.
- Global variables do not belong to any cluster or cluster level. The cluster invariant is modified to "if $t \rhd_c v$ and v is not global then t and v belong to the same cluster". The scheduler keeps a separate list of dirty constraints on global variables and always processes them last, when no dirty clusters are available.
- If $t \preccurlyeq_i u$ and t is global then the algorithm sets $t = u$ and deletes the instance constraint.

9.4. Exceptions

SEMI encodes exceptions thrown by methods as auxiliary result components of method types. In practice, as far as SEMI is concerned, any exception thrown by a method can be propagated to the top level . This means that variables corresponding to thrown exceptions (or their components) satisfy the same constraint property given above for variables corresponding to global data. Thus, using the "globalization" optimization on variables for thrown exceptions causes no loss of precision, and in practice the savings in space and time are significant.

10. A failed optimization: cut-throughs

10.1. Example

Consider the following program:

```
Foo f1() { return new Foo(); }
Foo f2() { return f1(); }
Foo f3() { return f2(); }
... f3() ...
```

Any necessary components of the new `Foo` will be propagated to the call site for `f3`. The variables corresponding to the results of `f2` and `f1` will also get copies of the components. This is unsatisfying because handling these semantically meaningless layers of abstraction could exact a significant cost in time and space for the solver.

10.2. Cut-throughs

I attempt to resolve this problem by introducing a notion of a "cut-through instance": a single instance constraint that summarizes a chain of instance constraints. In the example, a single cut-through instance could connect the result of "new `Foo`" with the result of `f3`. This meant that the components of the object need not be expanded in the results of `f2` and `f1`.

It was very difficult to implement. A large amount of bookkeeping was required to ensure consistency, and it was tricky to implement efficiently. To make the implementation tractable, I had to carefully restrict the circumstances in which cut-through edges could be used. Unfortunately, experiments showed that on real examples cut-through instances were hardly ever being used, so I disabled them. I do not recommend introducing this style of optimization.

11. Conclusion

I have presented a description of the strategies used in my implementation of SEMI, a solver of systems of polymorphism constraints. All of them, except for "cut-throughs", made significant contributions to the performance of the system working on real programs (e.g. Sun's `jar`, `javap` and `javac` tools, and Ajax itself).

I have certainly not exhausted the possibilities for improvements. For example, there seem to be further opportunities to reduce space by implicitly representing some instance and/or component constraints and reconstructing them on demand. Other future work might involve extending the constraint system with new information or new kinds of constraints, to provide better accuracy.

12. Bibliography

[DM82]	L. Damas and R. Milner, Principal Type Schemes for Functional Programs. *Proceedings of the Ninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1982, pp. 207-212.
[H93]	F. Henglein. Type inference with polymorphic recursion. *TOPLAS, Volume 15, No. 2, 1993.*
[O99]	R. O'Callahan. The Design of Program Analysis Services. CMU-CS-TR-99-135.

Appendix A: Polymorphic Recursion, Unrestricted Recursive Types and Principal Types

Consider a standard lambda language with a type system having polymorphic recursion and unrestricted (μ) recursive types. In such a language, there are typable program terms that do not have principal types. Note that in the setting of μ-recursive types, a type T for a term `f` is principal iff T is a type of `f` and every type of `f` is equivalent to an instance of T, where type equivalence means that the (possibly infinite) regular labelled trees corresponding to the types are identical.

Consider the following function, written in ML-like syntax:

```
fun f (a, b) = f b
```

This function is typable using polymorphic recursion and unrestricted recursive types, but there is no principal type. Here are some possible types (all free variables are assumed to be universally quantified):

- $(\mu t.\ v \times t) \rightarrow u$
- $w \times (\mu t.\ v \times t) \rightarrow u$
- $x \times (w \times (\mu t.\ v \times t)) \rightarrow u$

Informally we could write these types as "$(v, (v, (v, \ldots))) \rightarrow u$", "$(w, (v, (v, (v, \ldots)))) \rightarrow u$", and "$(x, (w, (v, (v, (v, \ldots))))) \rightarrow u$". The principal type would need to have an unbounded number of quantified variables, but such types do not exist.

More formally, suppose T is the principal type. Let m be the number of free variables in T. Define

$J_0 = \mu t.\ v \times t$

$J_n = w_n \times J_{n-1} \qquad (n > 0)$

For all n, $J_n \rightarrow u$ is a type of f. Therefore there is a substitution S such that S(T) is equivalent to $J_m \rightarrow u$. $J_m \rightarrow u$ has more free variables than T; therefore, there is a free variable of T (referred to as e) such that S maps e to a term equivalent to a subterm of $J_m \rightarrow u$ containing at least two free variables (I will refer to the latter subterm as the "expansion term"). These are the subterms of $J_m \rightarrow u$, modulo equivalence:

1. $J_m \rightarrow u$
2. u
3. J_i $(1 \leq \mathrm{i} \leq m)$
4. w_i $(1 \leq \mathrm{i} \leq m)$
5. v
6. $\mu t.\ v \times t$

Cases 2, 4, 5 and 6 do not contain at least two free variables, hence cannot be the expansion term. Case 1 cannot be the expansion term, for then $T = e$ which is not a type of f. Therefore the expansion term is J_i (for some i, $1 \leq \mathrm{i} \leq m$).

Let S' be the same substitution as S except that e is mapped to "int". S'(T) is equivalent to the tree for $J_m \rightarrow u$ with one or more subtrees equivalent to J_i replaced by "int". But since w_i occurs just once in the tree for "$J_m \rightarrow u$", there is only one such subtree — the actual occurrrence of J_i introduced by the production rules. Therefore $S'(T) = K_m \rightarrow u$ where

$K_i = \mathrm{int}$

$K_n = w_n \times K_{n-1}$ $\qquad (n > 0)$

It is easy to see that this is not a type of f, violating the assumption that T is a principal type.